# Visual Programming in Prolog

## Conference or Workshop Item

# Visual Programming in Prolog

Simon HOLLAND

Department of Computing Science
Kings College
University of Aberdeen
Aberdeen
Scotland AB9 2UB

Tel : 44 224 27 2284
Fax : 44 224 48 7048
email (Janet) : simon@uk.ac.abdn.cs

**ABSTRACT**

A new, simple, complete visual formalism for programming in Prolog is presented. The formalism is shown to be equivalent to the standard textual notation for Prolog. We demonstrate some kinds of relationships and styles of programming that appear to be particularly lucid for novices when presented in the graphic notation, while other aspects of Prolog are identified that are clearer in the standard notation. The symbolism is proposed as a specialised complement, but not a replacement, for the traditional notation. The design of a computer interface called VPP is presented that supports visual programming in Prolog using the graphical notation. We discuss a prototype of VPP that has been implemented. We give examples to demonstrate how VPP has the capacity to allow users with little or no knowledge of Prolog to explore, understand, modify and create Prolog programs used to represent domain expertise in intelligent tutoring systems and interactive learning environments. We argue that the use of VPP may have advantages for beginners in the early stages of learning to program in Prolog, and may help to avoid certain misconceptions. We informally analyse the structure and properties of the notation from an abstract human-machine interaction viewpoint. An extension of the interpreter (dubbed VPE) designed for visualising Prolog execution spaces is presented. The limitations and possibilities for further work of both systems are identified and discussed.

## 1 INTRODUCTION

This paper describes a system for visual programming in Prolog. We have dubbed this system VPP (short for Visual Programming in Prolog). The system allows users to edit and create Prolog programs graphically by moving about and connecting graphical symbols on screen. Queries can be constructed in a similar manner. We will also discuss an extension of VPP dubbed VPE (Visualiser for Prolog Execution), which animates a 3D model of program execution that employs exactly the same pictorial building blocks as VPP. A prototype of VPP has been implemented on a SparcStation. This allows users to construct programs graphically and translates them into conventional textual Prolog code. The VPE part of the system (for execution visualisation) has not yet been implemented.

## 2 MOTIVATION

Shu (1988) notes several motivations for designing visual programming languages. Some of the reasons can be summarised as follows:

- Pictures can show multiple relationships concisely and clearly.
- Visually encoded relationships can be easier to remember.
- Pictorial representations can provide an incentive for learning to program.

(Of course, it does not follow that any particular programming language will be effective - this must be tested empirically.) A further motivation, based on observations by Shniederman (1987), is that if important conceptual relations can be encoded in ways that make them perceptually salient, this can free up scarce cognitive resources to deal with higher level problems. Systems have been developed for visual programming in a wide range of programming paradigms, such as Levitt's "Hookup" for a simple dataflow language (Levitt, 1987), the MAX system for object-oriented programming (Puckette, 1988), Pascal-BLOX for Pascal (Glinert,1986), and Tinkertoy for the functional language Lisp (Edel,1986). Other notable visual languages include Mandala (Lanier, 1984). Surprisingly, no previous system for visual programming in Prolog (as opposed to execution visualisation - such as the Transparent Prolog Machine - Eisenstadt and Brayshaw, 1987) appears to have been designed or implemented to date. As we have already noted, the effectiveness of any *particular* visual programming language in terms of ease of learning, effectiveness of use, ease of debugging, and so forth, is a matter for empirical testing: we hope that VPP and VPE will soon be tested in this way. Meanwhile, it is worth mentioning just one theoretical motivation for developing a language for visual programming specifically in Prolog.

Designers of intelligent tutoring systems, expert systems, computer aided design systems, and so forth, sometimes need to provide users who are non-programmers with a simple way to understand and modify constraints or heuristics, etc. embedded in systems encoded in Prolog. Users may need to explore very general "what if" questions concerning complex AI models such as "How would this building / electronic circuit / experiment in physics / piece of music / etc. look if this design constraint were to be modified?". A system for visual programming in Prolog in conjunction with a suitable metaphor or "story" for novices specific to VPP (Holland, 1991) might make it possible for non-programmers to modify *certain styles of programs* in intuitively natural ways without having to learn Prolog at all.

Equally, a system for visual programming in Prolog could allow the construction of application-specific visual programming kits for music, animation, etc. analogous to Levitt's dataflow-oriented Hookup system or Puckette's object-oriented MAX. The chief benefit here of VPP is that Prolog has (nearly) the full expressive power of logic, and could allow complete beginners to exploit existing AI models of musical expertise and style (e.g. see Holland ,1989) or any other AI-encoded area.

At the same time, VPP might have applications in the early stages of learning to program in Prolog, especially in conjunction with the execution model VPE. There may even be one or two cases in which the graphical formalism may be able to give fresh insights to relatively experienced Prolog programmers(see section 7).

## 3 DIAGRAMING PROLOG PROGRAMS

We will now present a slightly idealised version of the notation and system by means of a set of elementary examples covering all of the key constructions of Prolog.


Fig. 1. A relation in VPP.

### 3.1 Relations

Terms (variables and constants) will be represented in VPP as links, and relations as nodes. So, for example, this gives the following visual representation (figure 1) in VPP for the fact:

parent(abe, ben).

An asymmetrical box shape has been used for the relation "parent " to emphasise the fact that the 'parent' relation is asymmetrical (i.e. not commutative). However, this is inessential: VPP works perfectly well (and in some senses better) if rectangular boxes are used uniformly for all predicates. Variables and constants are distinguished in VPP diagrams just as in conventional Prolog notation by whether their first character is lower or upper-case.

### 3.2 Shared terms

Where one or more clauses have the same constant term used in more than one place, VPP

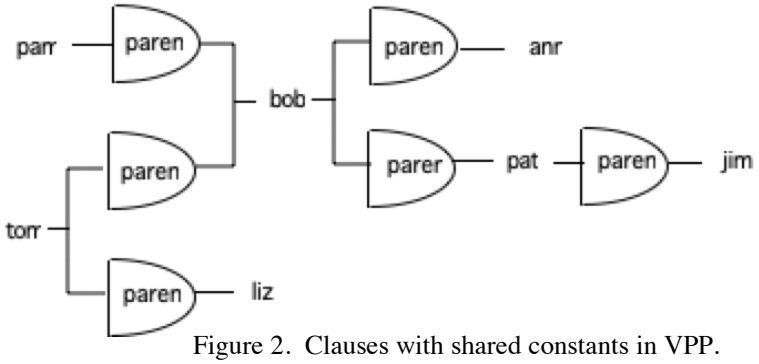


Figure 2. Clauses with shared constants in VPP.

optionally allows a single graphical instance of that term to serve for more than one clause. For example, consider the following set of clauses (borrowed from Bratko, 1990).

parent (pam,bob).
parent (tom,bob).
parent (tom, liz).
parent (bob, ann).
parent (bob, pat).
parent (pat, jim).

These can be represented in a VPP diagram as shown in figure 2. Notice that one can see at a glance here that bob is grandparent, pam is a great-grand parent and so forth. In some application areas, this can be a considerable advantage. Note that if two or more clauses in a program have the

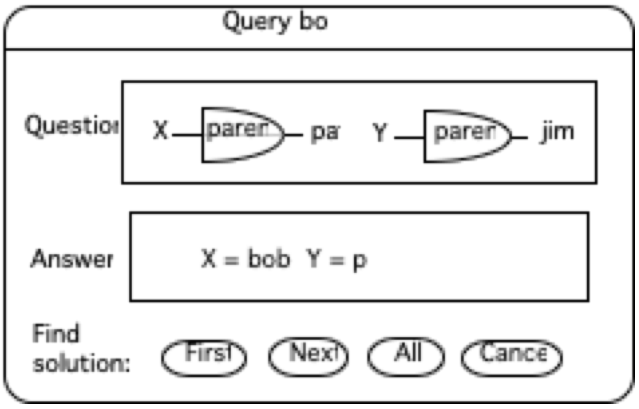


Figure 3. A conjunctive query.

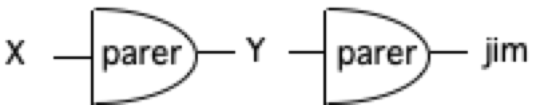


Figure 4. A conjunctive query with a shared variable.

same *variable* used in more than one place, and the user tries to join these up, VPP will not permit this unless the user will agree to encapsulate all of the clauses into a single rule (we will see how to do this in a moment) since Prolog does not allow conjunctive clauses in programs. Clauses in a Prolog database must have a well-defined order. The clauses in a VPP program are considered by default to be ordered from left to right, top to bottom. Clauses in the database can be reordered by dragging them around.[1]

**3.3 Queries**

VPP reserves an area of the screen (the query box - figure 3 ) in which queries can be posed graphically. To make the simple query *parent(bob, X)?*, the corresponding graphical construction is moved to, or assembled in, the query window and a solution requested by pressing the "Find solution" button. To perform a *conjunctive* query, more than one clause can be put into the query box at a time.

**3.3.1 Conjunctive queries with shared variables**

In contrast with the restriction on shared variables in *programs*, when one or more clauses in a conjunctive *query* have terms (constants or variables) in common, VPP optionally allows a single graphical instance of that variable to serve for more than one clause. For example, figure 4 is quite legal as a query.

**3.3.2 Disjunctive queries**

Disjunctive queries are expressed by posing the disjunctive clauses as queries one after another.

**3.4 Rules**

In VPP diagrams, the body of a rule is a graphically encapsulated inside a box representing the head. Lines representing arguments to the rule as a whole "stick out" from the outer box. So for example, the following rule:

[1] Dragging around boxes may not always be convenient. Optionally, VPP can be asked to show numerical ordering labels next to each clause (figure 8). By clicking on these labels and typing in new numbers, sections of the database can be re-ordered.

```
sister(X,Y):-
        parent (Z,X),
        parent(Z,Y),
        female (X),
        different(X,Y).
```

would be represented in VPP as in figure 5.
Where one or more goals in a rule have terms (variables or constants) in common, VPP optionally allows a single graphical instance of that term to serve for more than one goal. So, for example, in figure 5, the variables X, Y and Z only need to appear once each. As with clause order in a program, the goals within a rule body are considered to be ordered from left to right, top to bottom. Optionally, the order can be adjusted without moving goals around physically. VPP may be made to display numbers showing the order of each goal within the body(fig 5). The ordering of goals in the body may be changed by re-entering these numbers (VPP may complain if you take a number that is already taken).

**3.5 Structured terms**

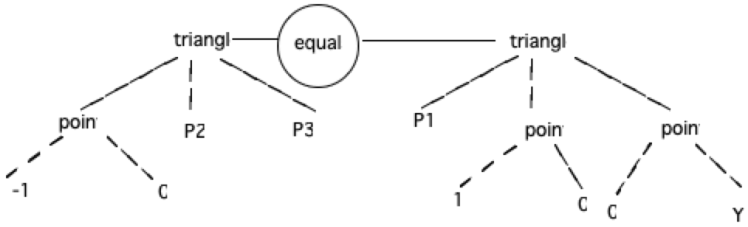


Figure 6. A relation between two structures.

Compound terms (structures) can be viewed as tree-structured variables. We need in our notation to distinguish between lines showing common occurrences of terms and lines showing tree structuring of variables. VPP adopts the convention that the nesting of components (functors and terms) within structures is shown by dotted lines. So, for example, the following clause with structured terms may be represented diagrammatically as shown in figure 6.

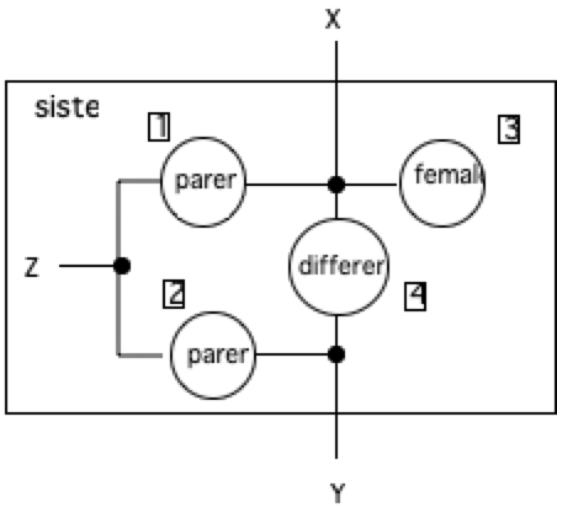


Figure 5. A rule.

equals(triangle(point(-1,0), P2, P3)), (triangle(P1, point(0,1), point(0,Y)).

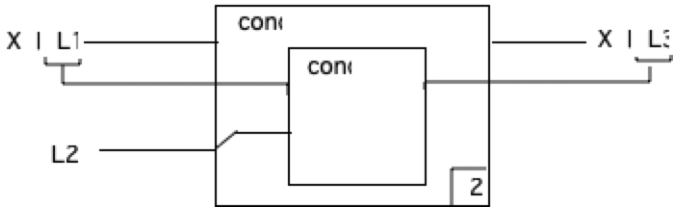


Figure 7. One of a number of alternative conventions
for showing shared terms in lists
in VPP.

Lists, which may be viewed as a special case of tree-structured variables, may be displayed either as tree structures using the dot functor, or using the conventional textual notation for lists (e.g. see figure 7). As ever, identical terms in compound structures which occur in more than one place in a collection of ground clauses or in more than one place within a given rule may be represented by a single graphical instance of that term (figure 7). However, this is optional and is not always recommended as it sometimes hinders clarity .

**4 Programming using VPP**

We will now give a slightly idealised account of the VPP programming environment. The user is presented with three main resources: a permanent menu or strip of graphical tools, a programming window and a query window. The most prominent tools are a soldering iron to connect up boxes, a pair of scissors to cut connections and a selection of unnamed boxes (figure 8).
The user chooses a box template from the tool strip with the appropriate number of arguments (seen as "docks" or nodes on the template). The typical activity for users is to pick up box templates with the appropriate numbers of "docks", place them in the program area and connect them up. Boxes may be subsequently moved around or deleted. Collections of boxes (with or without adjoining lines and terms) may be selectively "lassoed" together and moved about *en masse*. Boxes may be grown or shrunk to allow for movement of clauses in and out of rule constructions. Programs can be of indefinite size as the program area is scrollable. Views of a program can be given at various magnifications, and there are indexes and find functions to help move around in substantially sized programs. There is a tool to allow the user to type in names on boxes or to alter existing names. The same tool may be used to type names of terms onto connecting lines. Lines can be joined to argument "docks" on boxes, or joined to other

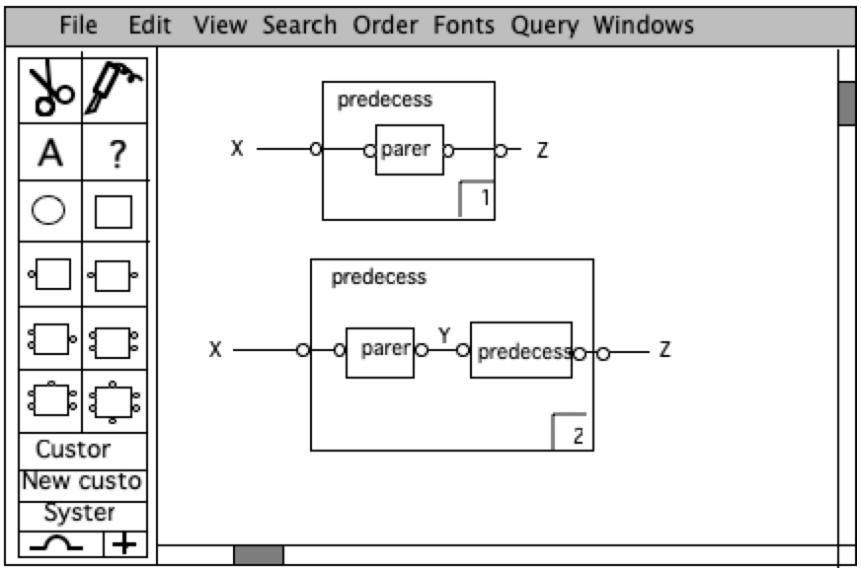


Figure 8. A slightly idealised view of the VPP environment.

lines (although sometimes VPP may complain if a connection gives rise to an inconsistency, and ask for a term or connection to be withdrawn). Graphical programs prepared earlier may be loaded and subsequently edited. VPP can be asked to generate textual code at any time, though it may complain about syntax errors in some cases. As far as possible, syntax errors have been designed out of the system: for example VPP will make up box names and term names if they have not been supplied. Textual Prolog code can be displayed in a separate window side by side with the graphic program. Optional tools include an ordering tool for numbering clauses within programs, goals within rules, and docks within a box. By default, dock positions are numbered clockwise from the 9 o'clock position. The positions of argument "docks" on boxes can optionally be moved around for convenience - for example

in the case of boxes with just one argument, it is sometimes useful to have the dock on one side, sometimes on another. The standard boxes are rectangular with from 0 to 10 docks, but "custom" box templates can easily be produced with any basic shape, name, number and orientation of docks and then installed for convenience on a menu. There is a "croquet hoop" tool to allow lines to pass over other lines without touching. There is an optional join tool to emphasise that two lines are joined together. There is a "constrain" tool to make it easy to draw horizontal, vertical and forty five degree lines. To make all these tools easy to use, "MacDraw-like" conventions are used.

**5 VISUALISING PROLOG EXECUTION**

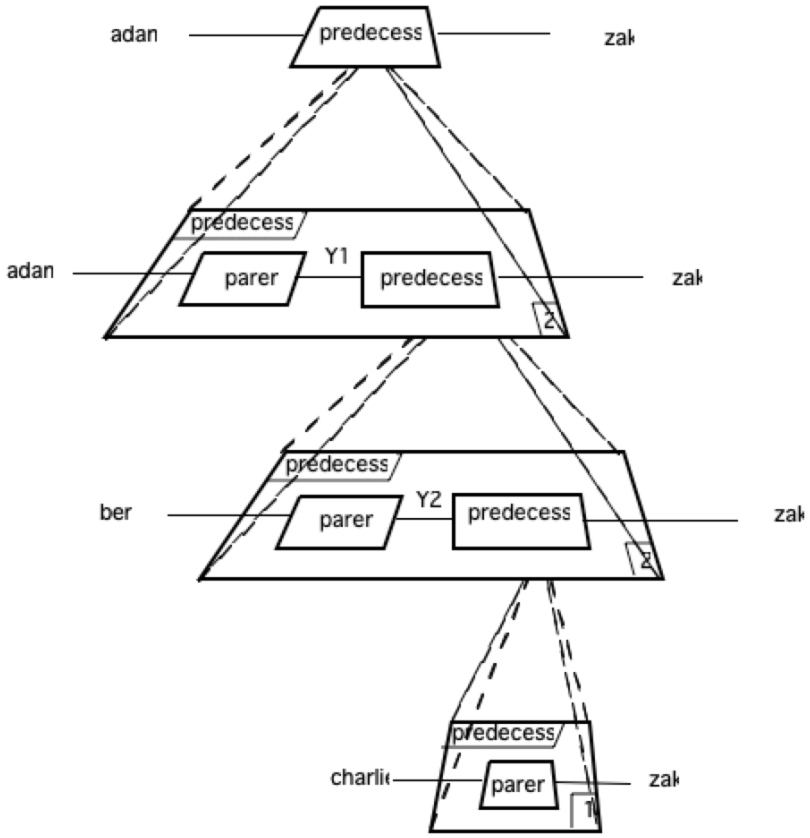


Figure 9. Execution of a simple recursive program seen in VPE.

To see how we might go about visualising the execution of a VPP program, let us consider a simple recursive example. Here is a simple recursive procedure for "predecessor".

```
predecessor(X,Z):-    parent(X,Z).
predecessor(X,Z):-
      parent(X,Y),
      predecessor (Y, Z).
```

This is represented graphically in figure 8. Let us imagine the following database of facts:

```
parent(adam, ben).
parent(ben, charlie).
parent(charlie, zak).
```

If we now pose the query:

```
predecessor(adam,zak)?
```

we can visualise the execution as a succession of internal queries in the database which are answered in each case by forming an "exploded diagram" of the query. By stacking corresponding VPP diagrams three dimensionally, we very naturally come to the visualisation shoen in figure 9.

Note how the unification of a goal with a clause is indicated using an "exploding diagram" graphical metaphor. This graphical effect, taken together with rule boxes that visually bind the body of a rule into a visual unity, correspond to the abstract notion of an AND tree. (The graphical equivalent of the "OR" element, corresponding to backtracking and choice of clauses will be dealt with in a moment). VPE diagrams can be shown in views that show more or less information. For example, it is possible to

show just the calling pattern (more or less as above), or to include the passing of values explicitly using arrows.

**5.1 HMI aspects of visualisation**

It is assumed that in a full implementation of VPE, the user would be able to examine the structure from different angles by flying around and through the trees, rotating, selectively pruning, and abstracting them. We will give some precedents to help justify this assumption in a moment, but here is general justification. AI researchers have a standard argument in which they point out that according to Moore's law (Moore, 1975), in five years from the present day, average users will have $2^5$ (or whatever) times as much RAM and MIPS as they do now. Therefore, it is argued, it makes sense for AI researchers to use top-end work stations of the present day to design applications that will trickle down to end-users five years hence. The analogous argument can be made for assuming very high quality graphical interface resources when carrying out long term interface design research. In fact, graphics capabilities in the marketplace tend to evolve more slowly than simple RAM size, etc, but now that GUI interfaces are finally sweeping even the conservative IBM-PC and Unix worlds, the argument has some force.

But why would we actually want 3D rotation and so forth? We believe that the standard interfaces of the future, as presaged by systems such as Xerox PARC's Information Visualiser (Clarkson, 1991), will have the facilities of machines such as current top-end, high-resolution, colour Silicon Graphics Iris machines (with 3D zoom, spin, pan, rotate, etc of rendered objects in hardware). Xerox PARC's Information Visualiser, designed for business users, uses visual structures such as 3D "cone-trees" and "cam-trees" for dealing with large hierarchical structures. The Information Visualiser has a battery of graphical techniques for displaying hierarchical, linear and unstructured information (much of it not normally considered to be three-dimensional) in various three-D presentations that exploit perceptual cues such as shadows on the "floor", brightness, size of nearer objects and so forth. One of the key sources of power is to convert higher level cognitive work into jobs that can be handled by the unconscious perceptual system. In this context, 3D rotation can be viewed as a way of presenting multiple relationships from different viewpoints while integrating them into a single higher conceptual unity. Combined with techniques to selectively suppress cluttering detail, this strategy has been found to be very successful (Clarkson 1991). Such techniques are so general and powerful, and exploit such strong underlying human resources that systems of this sort may ultimately become as pervasive as WIMP interfaces. Against this perspective, Visual Execution in Prolog may be seen as being a route to three dimensional visualisation of Logic Programming to fully exploit this new interface medium.

**5.2 Visualisation of backtracking.**

So far, we have not shown how backtracking is visualised (or how features like the cut and "not" work). In order to demonstrate these features and facilitate comparison, we have borrowed an example program from TPM (Eisenstadt and Brayshaw, 1987) which brings together in a very small compass many of the problems to be dealt with by an execution modeller. Here is the example program showing backtracking.

```
party(X):- happy(X), birthday(X).
party(X):- friends(X,Y), sad(Y).
happy(X):- hot, humid, not raining,!, swimming(X).
happy(X):- cloudy, watching_tv(X).
happy(X):- cloudy, having_fun(X).
cloudy.
hot.
humid.
having_fun(tom).
having_fun(sam).
swimming(john)
watching_tv(john).
sad(bill).
sam(sam).
birthday(tom)
birthday(sam)
friends(tom,john).
friends(tom,sam).
```

Figure 10. A simple example program reproduced from Eisenstadt and Brayshaw (1987).

Let us now consider the query *party(Name)?*. Figure 11 represents the execution of the program up to the first and only failure of birthday/1. Figure 12 shows the entire execution space of the program.

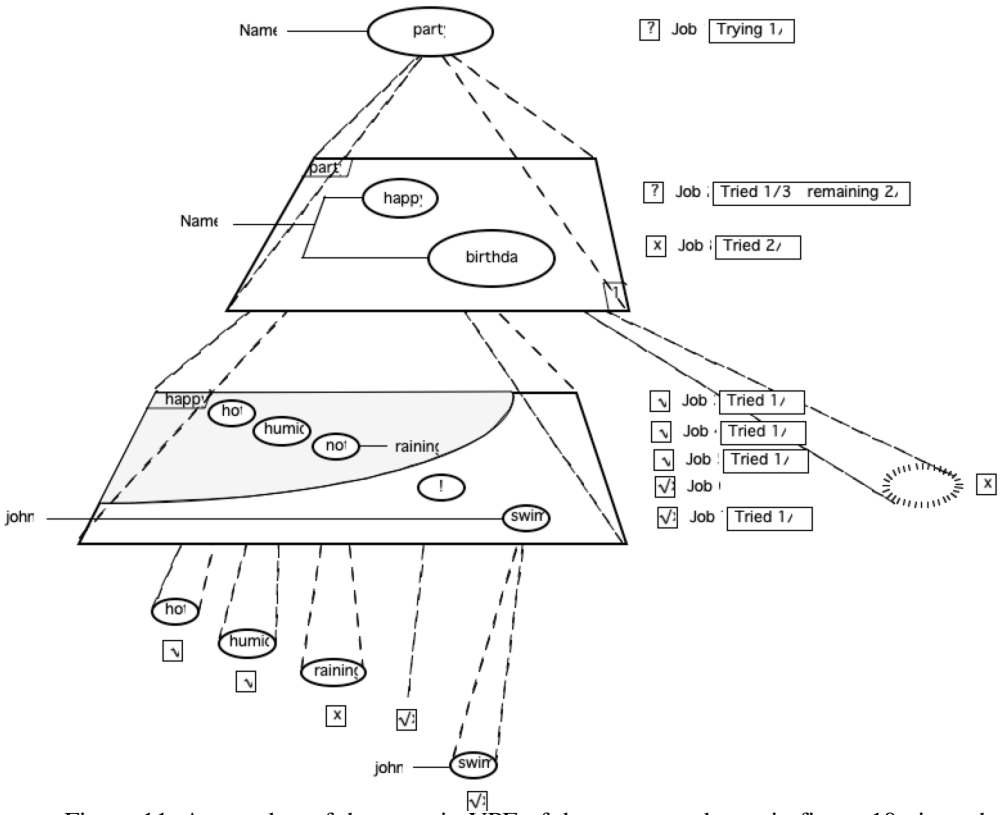


Figure 11. A snapshot of the trace in VPE of the program shown in figure 10 given the query *party(Name)?* up until backtracking begins with the failure of birthday/1.

**5.3 Reading a VPE diagram**

In the above diagram (figure 11), *'job-ticket's* are shown for each of the goals. These can be displayed in various degrees of detail or completely omitted. Representation of the "job-tickets" is related to the way the way that "procedure status information" is handled in TPM. The name "job-ticket" reflects the factory metaphor described in Holland (1991). Job ticket information is as follows. First of all, borrowing the convention from TPM, the *goal status box* shows whether a goal is being currently processed, whether it has succeeded, whether it has failed, or whether it initially succeeds and then fails (or is redone) on subsequent backtracking. The following symbols, respectively, are used (borrowed from TPM): question mark, tick, cross, tick/cross combination. The second symbol in the job-ticket is the *job-number:* every goal that is processed is given a job-number. Job numbers can be helpful for tracking the order in which work was done when looking at a trace, and helpful after backtracking to see what should be done next. Finally, the "progress report" shows how many clauses there are in the database to try for that goal, and how many have been tried so far. Note also that where goals fail to match a clause, this is shown with the "bursting bubble" sign and a cross signifying failure.

The effect of cut, using Eisenstadt and Brayshaw's excellent "matriarchy" terminology for describing the actions of a Prolog interpreter (Eisenstadt and Brayshaw, 1987), is to freeze into a non-backtrackable whole any of the cut's older sisters and their descendants, which is indicated by the frozen upper portion of the parent rule body in figure 11. The cut also eliminates any other possible "future stepfathers", hence the "remaining 2/3 cut" entry in the progress report for the remaining clauses of "happy" in figure 11. After a failure, the part of the computation between the failure and the point at which computation resumes in a forward direction is shown in ghostly grey to indicate backtracking (not shown here). The tree is continued by beginning a new exploded diagram to one side of, but at the same level as its previous incarnation, connected by a thick horizontal "Or" bar (note that because of our distinctive convention for displaying "and" trees, instances of "or" branches are easily distinguished from "and" branches).

**6 POSSIBLE BENEFITS OF VPP and VPE**

• In cases where non-programmers need to understand or modify Prolog code, but may not be interested in learning Prolog, VPP may score well. The "factory construction" metaphor outlined in Holland (1991) which gives a homely rationale for the execution of a Prolog interpreter may be useful in this connection.

• In databases and rules where there are many shared terms, VPP can allow inter-relationships to be noted rapidly without having to memorise variable names, scan for matches, so lessening load on short term memory.

• The depiction of recursion as a "Russian doll" style set of nested boxes (see figure 9) gives a visual analogue for recursion that, for some people, is very compelling and enlightening.

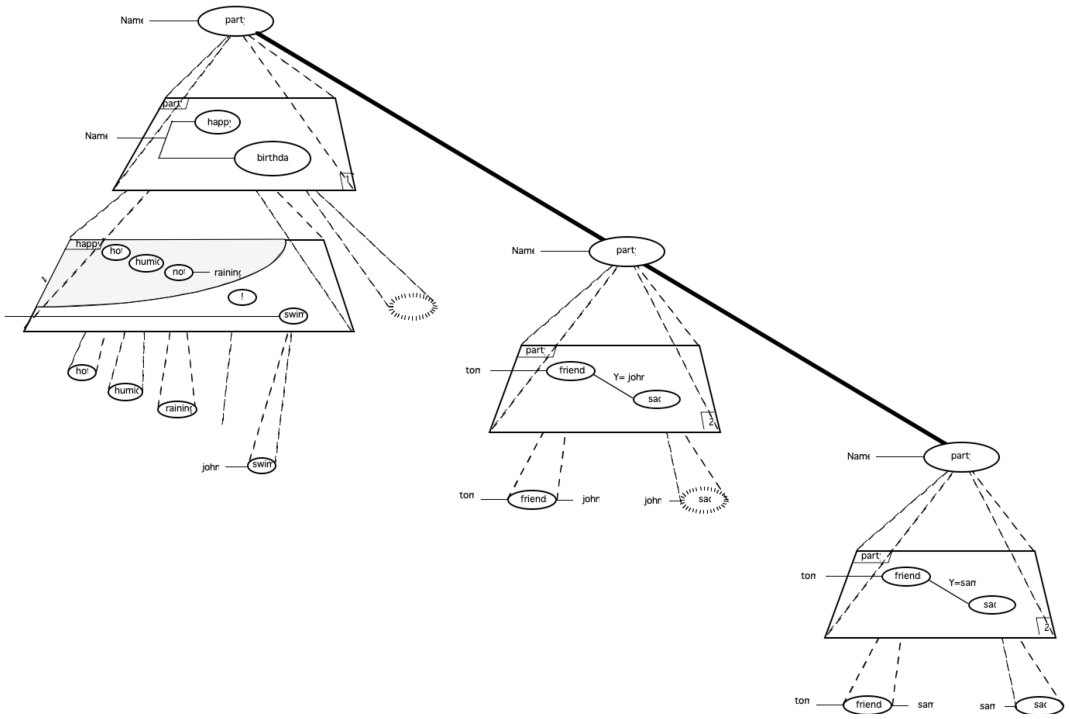

Figure 12. An outline trace in VPE of the complete execution space of the program shown in figure 10 given the query *party(Name)?*

• VPE may be useful for inspecting complex relationships broken down into their sub relationships (i.e. proof trees where branches that proved resulted in failure are ignored). Visual 3D presentation of this kind of information may be useful (for particular users and particular kinds of situation) for the reasons noted above and in previous sections.

• VPP makes it quite clear that predicates and structures are different kinds of objects. This may help to avoid a number of systematic misconceptions that some novices tend to acquire arising from this confusion.

• VPP can help to elucidate the tree-like nature of structures by showing their form graphically.

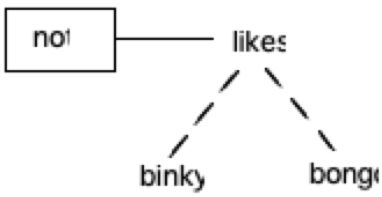


Figure 13. Representation of not(likes(binky,bongo).

• VPP can on occasion help to clarify aspects of Prolog even to slightly more experienced programmers. For example, it reveals graphically that the predicate 'not' could not take a predicate as an argument, but must take a term or structure (figure 13). This might help to avoid persistent misconceptions about "not" and the absence of second order behaviour in Prolog in general.

## 7 LIMITATIONS AND WEAKNESSES OF VPP

• VPP cannot always show up shared terms in lists very clearly

• The current implementation of VPP (in SunView on SunOS 4.03 on SparcStation 1) has many limitations, although all of the key capabilities are implemented. Limitations include lack of ability to edit clause and argument numbering, lack of group move facilities, lack of ability to edit dock positions, and so forth.

• VPE is not intended as a full-blown debugger for professional programmers. For that, see TPM . Various extensions or VPE have been designed which in principle could make it as fully-featured a debugger as TPM, although that is not its primary purpose.

## 8 RELATED IDEAS AND SYSTEMS

The core idea of VPP, and then VPE was inspired by Steele's notation for constraint programming (Steele, 1980) and by attempts to design a graphic programming language for a constraint-based musical planner (Holland, 1989). The most closely related existing system to VPE is Eisenstadt and

Brayshaw's Transparent Prolog Machine. VPP provides both a way of programming in Prolog and visualising program execution graphically using a single graphic metaphor, unlike TPM which is an execution model only. On the other hand, TPM is optimised as a debugger, and its relatively abstract notation may be an advantage for expert programmers for this purpose.

**9 FURTHER WORK**

Many possibilities for further work have already been noted, for example, VPP would benefit from a fuller and more flexible implementation. Trials with users are eagerly anticipated, even using the current simple prototype. It would be interesting trying to implement VPP "in reverse" i.e. a program that read in Prolog in the standard notation and attempted to plan out a clear, well-laid out version in VPP. This would require the encoding of expert layout heuristics. Some aspects of VLSI routing algorithms might be exploitable in this context. Note that from the perspective of Wenger (1987), VPP may be viewed as an interface for Prolog with *full epistemic fidelity,* and with the potential to give any system coded in Prolog full epistemic fidelity. Extending this idea, VPP could be used in virtual reality systems (Edwards and Holland, forthcoming) as a natural set of building blocks for manually assembling fully working logic programs via a homely metaphor(Holland, 1991).

**10 CONCLUSIONS**

We have demonstrated a new, simple, complete visual formalism for programming in Prolog. The design of an implemented computer interface for visual programming in Prolog using the notation has been outlined. An extension of the interpreter (dubbed VPE) designed for visualising Prolog execution spaces that employs exactly the same pictorial building blocks has been presented. We have noted some kinds of relationships and structures that may be particularly lucid for novices when presented in the graphic notation. The system may have particular benefits for building domain-specific graphic "construction kits". Links are noted to recent developments in three dimensional visualisation systems at Xerox PARC and elsewhere.

**ACKNOWLEDGEMENTS**

Grateful acknowledgement is made to David Philip for his work in implementing the prototype of VPP.

## REFERENCES

Bratko, I. (1990) *Prolog Programming for Artificial Intelligence*. Menlo Park, Addison Wesley.

Clarkson, M.A. (1991) An Easier Interface. *Byte*. Vol. 16 No. 2 February 1991.

Edel, M. (1986) The Tinkertoy graphical programming environment. *Proceedings of IEEE 1986 Compac* Oct 1986, pp. 466-471.

Edwards, A. and Holland,S. (forthcoming) Eds. *Multimedia and multimodal interface design in education*. Springer Verlag, London.

Eisenstadt, M. and Brayshaw, M. (1987) An integrated textbook, video and software environment for novice and expert Prolog programmers. *Proceedings of the 2nd International Conference of the Prolog Education Group (PEG 87),* Exeter, UK 8th-10th July 1987.

Glinert, E.P. (1986) Towards 'Second Generation' Interactive graphical programming environments. *IEEE Workshop on Visual Languages* (June 1986) pp. 61-70.

Holland, S. (1991) Extended Preliminary report on VPP: a system for visual programming in Prolog. Technical report, Department of Computing Science, University of Aberdeen.

Holland, S. (1989) Artificial Intelligence, Education and Music. Doctoral dissertation (CITE technical report no. 88) Open University, Milton Keynes.

Lanier, J. (1984) Cover, *Scientific American* Sept 1984.

Levitt, D. A, Burt Sloane, Mike Travers (1987). Hookup! V0.80 Computer software for the Macintosh, Media Lab MIT.

Moore, G.E. (1975) Progress in Digital Integrated Electronics, *Proceedings IEEE Digital Integrated Electronic Device Meeting,* 1975, page 11.

Puckette, M. (1988) The Patcher. *Proceedings of the International Computer Music conference 1988*. Feedback papers 33, Feedback Studio Verlag, Cologne.

Shniederman, B. (1987) *Designing the User Interface*. Menlo Park, Addison Wesley.

Shu, N.C. (1988) *Visual Programming*. New York, Van Nostrand Reinhold Company.

Steele, G.L. (1980) *The definition and Implementation of a computer programming language based on Constraints*. AI Lab, MIT (Doctoral Dissertation).

Wenger, E. (1987) Artificial Intelligence and Tutoring Systems. Morgan Kaufmann Publishers Inc., Los Altos, California.